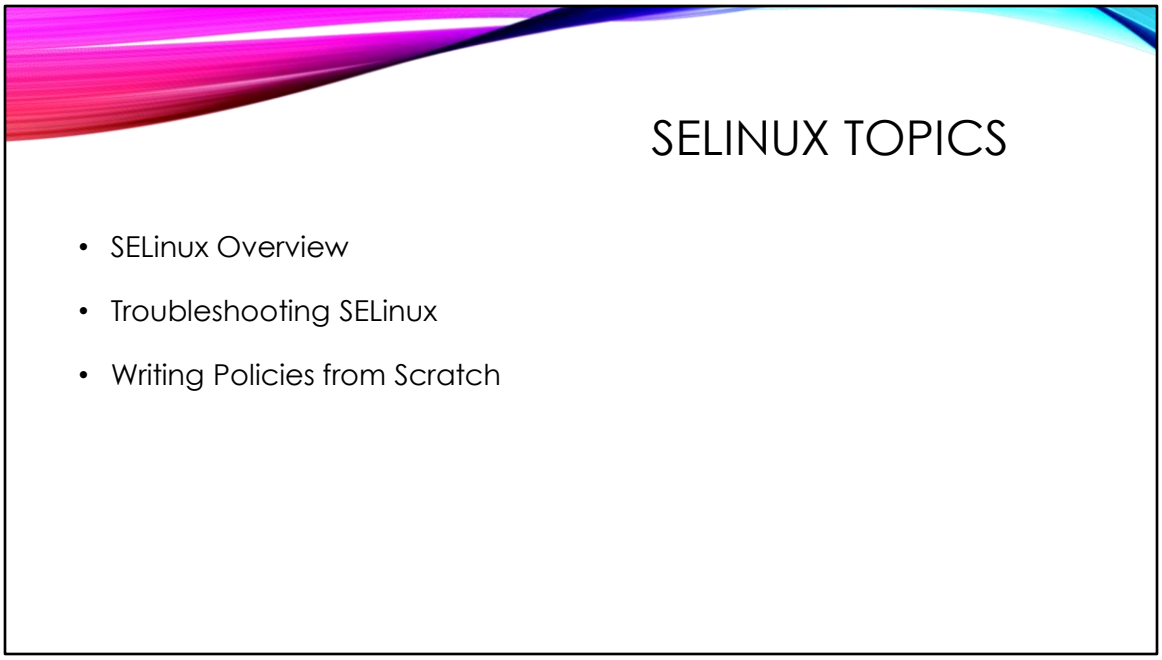
Download updates from *https://archive.org/details/HalSELinux*

Please support continued development of this material by taking one of my training classes or donating (US$20 is suggested) via PayPal (*paypal.me/halpomeranz*) or Patreon (*patreon.com/halpomeranz*).

Hal Pomeranz
hrpomeranz@gmail.com
@hal_pomeranz



The *SELinux Overview* section is a quick, high-level introduction to SELinux and some of the outermost administrative interfaces to the SELinux environment.

Next, we'll look at common issues and fixes when *Troubleshooting SELinux* – using some web server configuration examples.

While the first two sections contain the most critical material to survive and thrive in an SELinux environment, in the *Writing Policies from Scratch* section, we'll actually work through a complete example of creating a brand-new SELinux policy for a real network service. This will give you some deeper insights into how SELinux works under the covers.

First, a general overview and high-level introduction to SELinux…

SELinux is a very fine-grained set of access controls implemented in a kernel module. It actually covers several different types of functionality:

Multi-Level Security (MLS)/Multi-Category Security (MCS)—These sorts of security controls are typically required at high-security sites that need to rigidly partition different categories of data (unclassified, classified, secret, top secret, etc.) on their networks. Outside of these kinds of environments, however, this level of access control is not widely used or needed. Luckily, you can simply ignore this functionality in a typical SELinux configuration by simply not using it—all objects will be placed in the same minimum-security level by default and can freely interact with each other.

Role-Based Access Control (RBAC)—One historic problem with the Unix security model has been "all or nothing" administrative access via the root account. RBAC is an attempt to address this deficiency by allowing sites to assign specific aspects of administrative privilege to a number of different roles. For example, you might have a "network administrator" role that allows somebody to configure the network interfaces on the device, while the "printer admin" role allows somebody control the print queues. The end game would be to completely disable the root account and just assign people to specific roles based on their job function.

However, there are a couple of issues:

- Nobody has actually published a meaningful RBAC policy for a typical enterprise Linux environment—even a decent subset of one—and creating one from scratch is an enormous effort. Even if somebody did create and publish a straw-man policy, it might end up being too site-specific to be useful to other organizations.

- While different flavors of Unix support RBAC in their kernels, every vendor's implementation is completely different from a configuration syntax and administrative implementation perspective. So, if you spend a lot of time crafting an SELinux-based RBAC policy, you'd have to spend a bunch more time "porting" that policy to your Solaris, AIX, etc. systems.

This is why `sudo` still tends to be quite popular. It gives you 80% of the fine-grained access controls you'd get from RBAC, and it's both easier to configure and portable across multiple Unix variants.

Type Enforcement (TE)—Type Enforcement is essentially an application "whitelisting" facility. Type Enforcement policies attempt to specify exactly which components of the operating system (files and directories, devices, sockets, etc.) a given application needs to interact with and what level of access the application needs (read access, write access, etc.). If an application attempts to stray outside the bounds of the defined policy, the kernel refuses to grant access to the requested resource. More often than not, this will cause the application to fail.

The goal is to prevent an application that has been compromised by an attacker from misbehaving in ways that would allow the attacker to compromise the larger systems. So, Type Enforcement is really an "application isolation" strategy, similar to `chroot()`. However, where `chroot()` can really only control the application by limiting to a particular subset of the filesystem, Type Enforcement can control many other aspects of the application's behavior (access to network sockets, for example, or finer-grained access controls to specific files) and still allow the application to run in the default OS directory structure (meaning no need to set up a `chroot()` directory with copies of binaries, libraries, etc.). You could even run an application with a combination of `chroot()` and Type Enforcement restrictions, though most organizations see Type Enforcement functionality as a superset of the security controls they get in a typical `chroot()` environment.

These days, most sites are at most adopting Type Enforcement only and largely ignoring both MLS/MCS and RBAC. The most popular Type Enforcement strategy is the so-called "targeted policy" implemented by default in Red Hat Enterprise Linux (and also used by Gentoo Linux). As of RHEL5, the targeted policy covers nearly all standard OS daemons that can be accessed from outside the box and a number of other programs as well, while leaving normal user and internal administrative processes alone (thus reducing the pain of implementing SELinux in most environments).

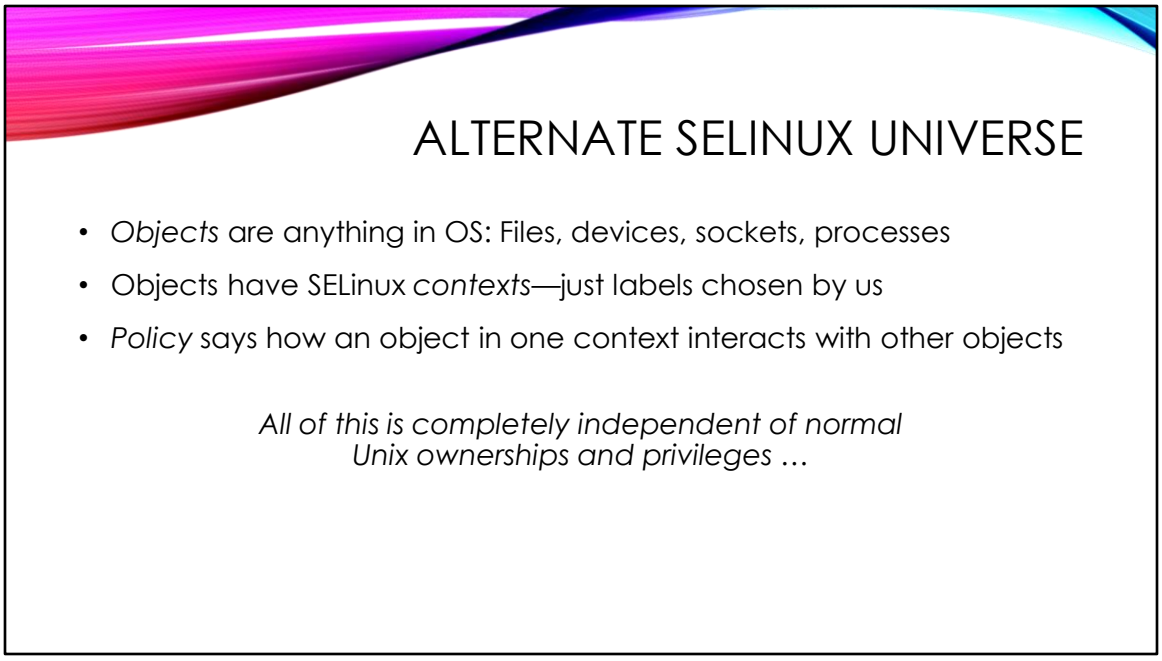
The original [SELinux] policy published at the NSA was the strict policy. Its goal was to lock down the entire operating system, controlling not only the daemons that live in system space but controlling the user space as well. Strict policy … adds the largest burden on users …

During the development of Fedora Core 2, we attempted to use strict policy as the default policy. We had multiple problems with this because the strict policy was governing the way users were running their systems. We had to cover all possible ways that a user would be able to setup their system. As you can imagine, we had a ton of problems and bug reports. Most people, when confronted with SELinux at that time, just wanted it turned off…

After our experiences with the strict policy, we went back and reflected on what our goals were. We wanted a system where the user was protected from System applications that were listening on the network.

These applications were the doors and windows where the hackers would enter the system. So we decided to target certain domains and lock them down while continuing to leave user space to run in an unconfined nature. Targeted policy was born …

From early Red Hat documentation



When SELinux talks about *objects*, it just means something in the operating system—whether that's a file, a directory, a device, a network socket, a process, or what have you. Each object is assigned a label that describes the *context* associated with that object. These context labels are just a set of conventions established by the folks who developed the SELinux policy on your system. When you're developing your own policy for new applications, you also end up creating new labels to identify the pieces of the operating system that are specific to your application.

SELinux policy rules control how an application running in a particular context, such as Apache web server running in the "`httpd_t`" context, can interact with other objects in the operating system—files in the web docroot, for example, which have context type "`httpd_sys_content_t`" in the default targeted policy. A policy statement might say that processes in the `httpd_t` context may read files that have `httpd_sys_content_t` context. Anything not specifically permitted by the SELinux policy is denied, so if the policy only grants the web server read access to files in the docroot, then if the process tried to overwrite one of those files (web defacement), the kernel would block the write attempt, returning an error code to the process. Depending on how the application is coded, the process may catch the error and keep running, or terminate and give up.

It's important to understand that SELinux essentially exists as an "alternate universe" from the normal Unix ownership and permissions rules that you're used to. That being said, it's important to maintain good discipline around normal Unix ownerships and permissions, because, again, most sites only implement SELinux Type Enforcement on certain critical processes and rely on normal Unix permissions to control access for interactive user sessions.

"-Z" SHOWS CONTEXTS

```
# ps -eZ | grep httpd
system_u:system_r:httpd_t:…  2773 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2775 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2776 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2777 ?  00:00:00 httpd
system_u:system_r:httpd_t:…  2778 ?  00:00:00 httpd
…
# ls -dZ /var/www/html
d… system_u:object_r:httpd_sys_content_t /var/www/html
# id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

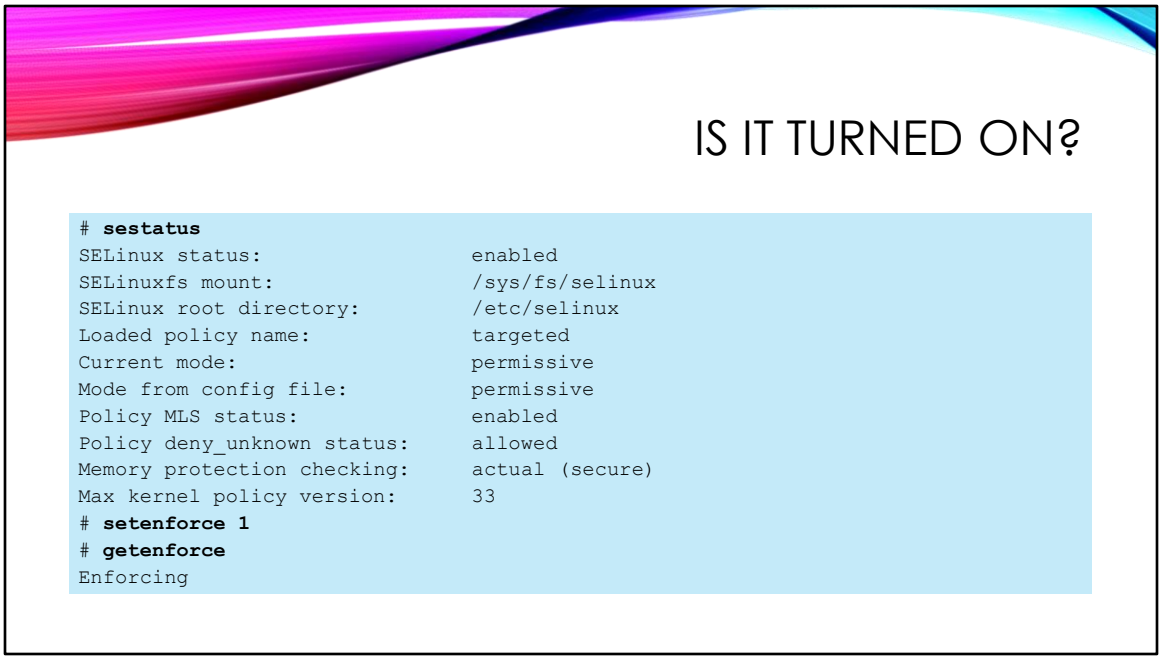
On a system that includes SELinux functionality, many of the standard OS programs have an added "`-Z`" flag, which displays the SELinux context information.

The context label is actually a triple of "user", "role", and "type." But if you're not using RBAC, "user" and "role" generally don't matter. The "user" field tends to be either "`system_u`" (for system processes and other objects in the file system) or "`user_u`" (for normal users and their processes). Processes and users tend to get shoved into the role "`system_r`" by default, while static objects like files in the filesystem are labeled as "`object_r`" in general. It's the "type" field that's really used to discriminate between objects in the typical Type Enforcement targeted policy. Note that the `*_u`, `*_r`, and `*_t` suffixes are purely conventional—they're just put there to make it easier to interpret the labels.

You'll also often see a string like "`s0-s0:c0.c1023`" appended to each context. This is the MLS/MCS label for each object. The first part is a range of "sensitivity levels" (think *confidential*, *secret*, and so on) that can see the object—by default all objects in the OS are shipped at sensitivity level zero (the lowest level) so that they can be used by all users on the system. The "`c0.c1023`" is a range of "categories"—think "proprietary" and so on—which, when combined with the sensitivity label, lets you say things like "*confidential and proprietary*" in you MLS/MCS policy. Again, the default here is "`c0.c1023`", covering the entire range of possible values. So, once again, any object on the system can interact with any other object without interference from the MLS/MCS policy.

Labels are inherited. If the `httpd` process runs a CGI script, that script will run in the `httpd_t` context unless the SELinux policy explicitly says otherwise (this is called a *transition* and does happen, as for example when `init` starts up a new daemon that wants to run in its own private context). Similarly, if you create a new file in a directory, it will tend to inherit the context of the parent directory by default.

You'll notice that our user session as reported by "`id -Z`" is labeled with "`unconfined_t`". The targeted policy has a generic rule that doesn't put any restrictions on objects in the `unconfined_t`. Effectively these objects, which include all user sessions and the processes they spawn, will not be constrained by SELinux at all, though they are still subject to normal Unix ownership and permission restrictions.



For most people, the only question they have about SELinux is, "Is that (expletive deleted) SELinux running?" Actually, SELinux cannot only be enabled or entirely disabled, but also if enabled you have the choice between *permissive* and *enforcing* mode. *Permissive* means that the kernel will log violations of the SELinux policy for the machine but still allow the application to continue. Typically, this mode is used for testing and development of new policy. If the system is in *enforcing* mode, then the kernel will terminate applications that violate policy.

If you plan to disable SELinux in your environment, please use *Permissive* mode rather than completely disabling SELinux. This way SELinux will log when the policy is being violated. These logs will be helpful for post-incident forensics when determining exactly what happened on your system.

`sestatus` will tell you whether or not SELinux is enabled and what mode it's running in. You can switch between permissive and enforcing mode with the `setenforce` command: Use "`setenforce 0`" (or "`setenforce Permissive`") to set permissive mode, "`setenforce 1`" (aka "`setenforce Enforcing`") turns on enforcing mode. `getenforce` will tell you which mode you're in.

Note that changes you make with `setenforce` will not persist across reboots. To make persistent changes, edit `/etc/sysconfig/selinux`, where you can choose to enable or disable SELinux and what mode to run in by default. If you're planning on changing between enabled and disabled states, you have to do it via a reboot.

SELINUX BOOLEANS

```
# getsebool -a | grep http
…
httpd_dontaudit_search_dirs --> off
httpd_enable_cgi --> on
httpd_enable_ftp_server --> off
httpd_enable_homedirs --> off
httpd_execmem --> off
httpd_graceful_shutdown --> off
…
# setsebool -P httpd_enable_cgi off
# getsebool httpd_enable_cgi
httpd_enable_cgi --> off
```

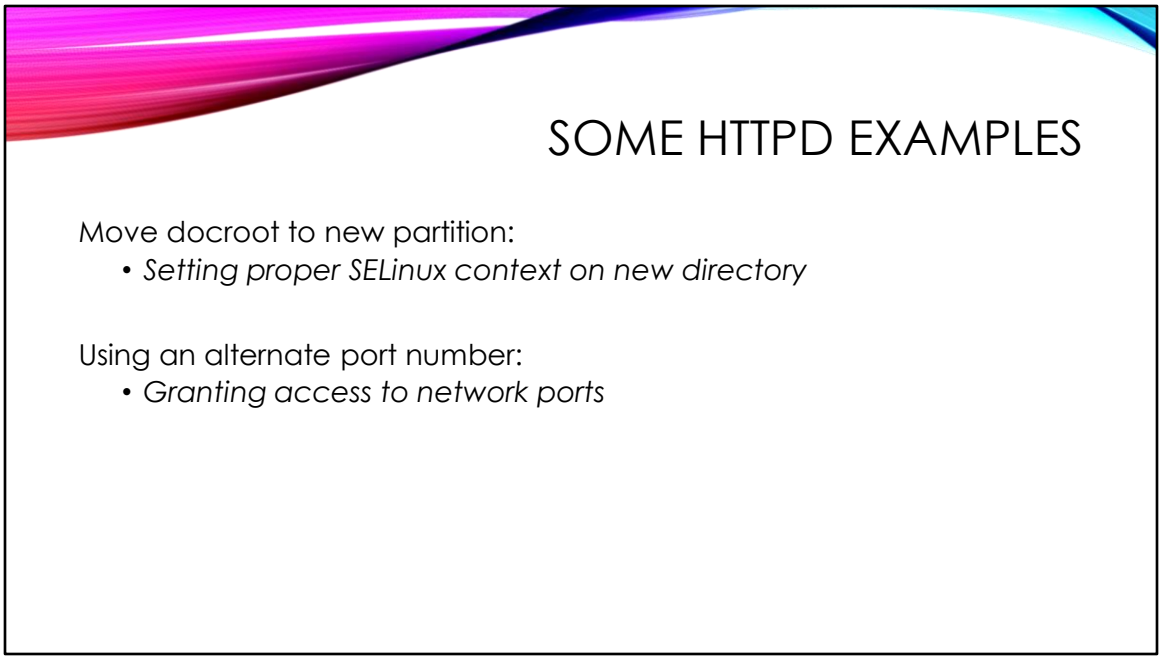
When enabled, the standard targeted SELinux policy is enabled by default. However, the policy modules for many applications include optional functionality that can easily be enabled or disabled via SELinux *booleans*—basically, simply on/off switches for the various bits of functionality.

You can get a list of all possible booleans and their current settings with "`getsebool -a`", or you can specify a single boolean to query as shown in the final example above. As you might expect, `setsebool` allows you to turn a given boolean on or off (1 for on, 0 for off). Note, however, that the changes you make with `setsebool` will not persist across reboots: Use "`setsebool -P …`" to make persistent changes.

In most cases, the name of the boolean pretty well describes what functionality it enables, particularly if you're familiar with the normal functioning of the application in question. However, if you're curious about exactly what each boolean enables, your only recourse is to get the source RPM for the SELinux policy for your system and look at the source code (the actual running policy files are stored in a binary format under `/etc/selinux` and are not human readable or easily reverse-engineered). Obviously, this is not an ideal method of documentation. More information on obtaining the SELinux policy source RPMs will be provided a little bit later in this module.

In the targeted policy that was implemented in RHEL5, each application had a `*_disable_trans` boolean associated with it. The idea was that when you turn these booleans on, it was supposed to allow the application to violate the SELinux policy, even when the system as a whole was in enforcing mode. Thus, they were supposed to be a temporary escape mechanism to test new functionality or quickly work around SELinux-related problems in production. Unfortunately, in practice, the `*_disable_trans` booleans didn't really end up working. In fact, these booleans have been completely removed from recent versions of the targeted policy and you won't find them at all in RHEL6 and later.

Now that we've oriented ourselves a bit in this new SELinux universe, let's look at some of the kinds of problems you will run into when you start using SELinux. This will also allow us to introduce some new commands for controlling objects in an SELinux environment.



While the default SELinux policy works well with the default configuration as provided by your OS vendor, the minute you start changing things, you start creating SELinux policy violations. On enforcing systems, this means your application gets terminated by the kernel. At this point, most people give up and disable SELinux because they don't know how to troubleshoot and fix the problem. So, let's work through some common kinds of failures.

The first kind of failure happens when you try to use a different directory configuration than the default—like when you try to get your web server to use an alternate document root. The normal problem here is that the alternate directories you're using don't have the proper SELinux context labels on them, so access is not permitted by the standard policy. The fix is to put the right labels on the new directories, and I'll show you how to do that.

Another common failure mode happens when you try to use an alternate port number for a service—for example, trying to get your web server to use an alternate port like 8888/tcp. Just like files and directories, sockets have contexts as well and you have to make sure the port you're planning on using is associated with the proper context. So, we'll also get to see how to manage contexts on port objects.

However, there are some problems that are not easily solved simply by relabeling. In these cases, we actually have to extend the default policy with our own rules. We'll look at these tools in the final section when we look at creating our own policies from scratch.

DOCROOT OF DOOM!

The scenario:

- You decided to create `/docroot` dir
- Updated `httpd.conf`
- Added some content

Mysterious errors when accessing content

Must be an SELinux problem, right?

So, suppose you decided to create a new `/docroot` directory to hold your web content. After creating the directory, you update your `httpd.conf` file appropriately and restart the server. The web server starts fine, but when you go to access any document under `/docroot`, you are forbidden to access the content.

```
# wget http://localhost/index.txt
--2021-07-09 14:10:01--  http://localhost/index.txt
Resolving localhost (localhost)... ::1, 127.0.0.1
Connecting to localhost (localhost)|::1|:80... connected.
HTTP request sent, awaiting response... 403 Forbidden
2021-07-09 14:10:01 ERROR 403: Forbidden.
```

Now you're a little crazed because you can do an "`ls -ld /docroot`" and see pretty clearly that it's a directory and the permissions are set correctly. And all the files under `/docroot` are world-readable.

```
# ls -ld /docroot /docroot/index.txt
drwxr-xr-x. 2 root root 23 Jul  7 09:00 /docroot
-rw-r--r--. 1 root root 12 Jul  7 09:00 /docroot/index.txt
```

You're pretty sure this problem must have something to do with that wacky SELinux thing, but how can you be sure?

IS IT REALLY SELINUX?

```
# ausearch -f index.txt
----
time->Wed Jul  7 09:05:13 2021
…
type=AVC msg=audit(1625663113.430:2787): avc:  denied  { getattr }
  for  pid=287174 comm="httpd" path="/docroot/index.txt" dev="dm-0"
  ino=35298437 scontext=system_u:system_r:httpd_t:s0
  tcontext=unconfined_u:object_r:default_t:s0 tclass=file permissive=0
```

Seems like it could be an SELinux issue…

SELinux uses `auditd` for logging events. That means we can use `ausearch` to find out if we have any events related to the `index.txt` file we asked for:

```
# ausearch -f index.txt
…
----
time->Fri Jul  9 14:10:01 2021
…
type=AVC msg=audit(1625854201.860:264): avc:  denied  { getattr
} for  pid=3589 comm="httpd" path="/docroot/index.txt" dev="dm-
0" ino=35298437 scontext=system_u:system_r:httpd_t:s0
tcontext=unconfined_u:object_r:default_t:s0 tclass=file
permissive=0
```

Obviously, these audit logs are not designed with readability as the foremost goal. But you can puzzle out what's going on here once you've been exposed to the madness for long enough:

1. First, we see that access is being "`denied`" to "`getattr`". "`getattr`" is short for "get attributes", and corresponds to trying to `stat()` a file or directory to figure out whether it exists and/or what kind of object it is.
2. The process triggering the denial is "`httpd`" (pid `3589`).
3. It's trying to access "`/docroot/index.txt`" (inode `35298437` on `/dev/dm-0`) which is a file ("`tclass=file`").
4. The context of the `httpd` process (the source context, "`scontext`") is `…:httpd_t` and the context of `/docroot` (the target context, "`tcontext`") is `…:default_t`.

Other `ausearch` options that might be useful in these cases include:

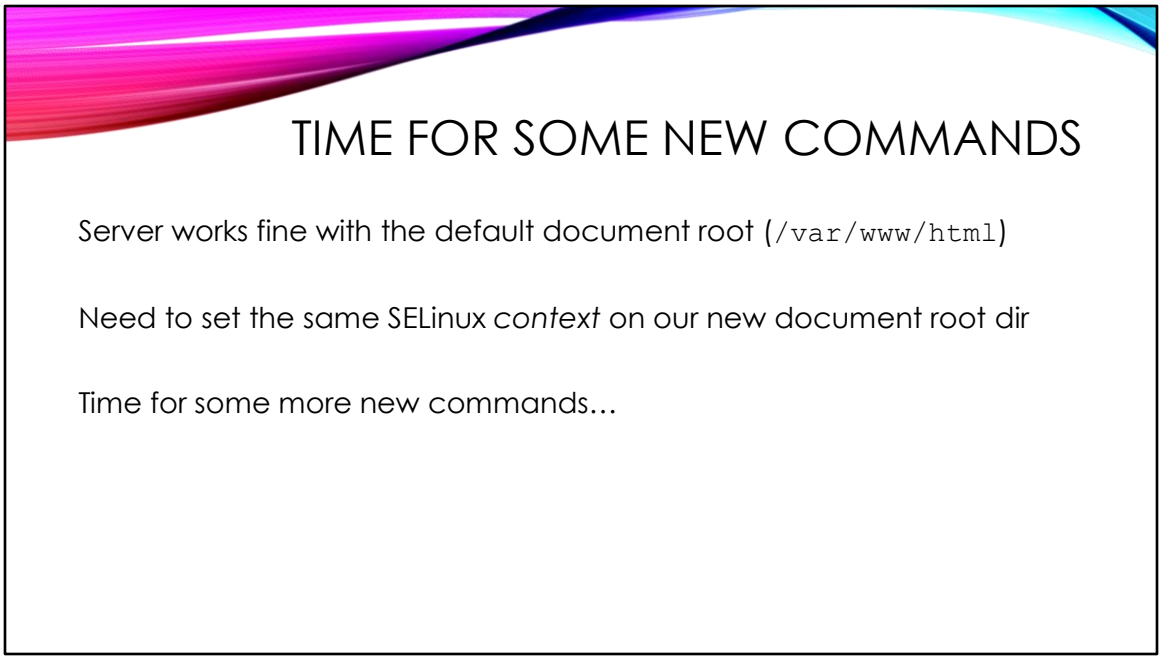
```
ausearch -c httpd      # search for entries where the "command" matches httpd
ausearch -se httpd     # search for entries where the SELinux context matches httpd
ausearch -m AVC        # search for SELinux alerts with "type=AVC"
```

`aureport -a` can also be useful to see a summary of `type=AVC` alerts. You can then use `ausearch -a` to get more details about specific events:

```
# aureport -a

AVC Report
============================================================
# date time comm subj syscall class permission obj event
============================================================
…
158. 07/09/21 14:09:30 ? system_u:system_r:system_dbusd_t:s0-
s0:c0.c1023 0 (null) (null) (null) unset 262
159. 07/09/21 14:10:01 httpd system_u:system_r:httpd_t:s0 4 file
getattr unconfined_u:object_r:default_t:s0 denied 263
160. 07/09/21 14:10:01 httpd system_u:system_r:httpd_t:s0 6 file
getattr unconfined_u:object_r:default_t:s0 denied 264

# ausearch -a 264
…
----
time->Fri Jul  9 14:10:01 2021
…
type=AVC msg=audit(1625854201.860:264): avc:  denied  { getattr } for
pid=3589 comm="httpd" path="/docroot/index.txt" dev="dm-0"
ino=35298437 scontext=system_u:system_r:httpd_t:s0
tcontext=unconfined_u:object_r:default_t:s0 tclass=file permissive=0
```



We know that the SELinux policy allows Apache to operate fine with the default document root—`/var/www/html` on Red Hat systems. That implies that if we could just copy the context that's set by default on `/var/www/html` and apply it to our new document root, then things would probably start working.

We've already seen "`ls -Z`" to display the current context of a file or directory. But now we need to learn how to change context labels on filesystem objects…

SETTING FILE/DIRECTORY CONTEXTS

Two-step process:

- First, use "`semanage fcontext ...`" to update policy
- Then "`restorecon -FR ...`" to fix contexts

Files created later automatically inherit context of parent dir

Let's start by using "`ls -Z`" to list the contexts on `/docroot` and `/var/www/html`. Actually, we already know the context on `/docroot` because it was the "target context" in the `audit.log` output, but let's see it again side-by-side:

```
# ls -Zd /var/www/html /docroot
d… root unconfined_u:object_r:default_t:s0 /docroot
d… root system_u:object_r:httpd_sys_content_t:s0 /var/www/html
```

OK, we need to set the context "`system_u:object_r:httpd_sys_content_t`" on `/docroot` and all the files underneath that directory. Just like `chown` and `chmod`, there's a `chcon` command for changing SELinux contexts on files and directories. However, the changes you make with `chcon` are not retained outside of the context labels on the files and directories themselves. If you have to restore your filesystem because your disk burns up, the changes you make with `chcon` are not saved anywhere. That means you have to manually go back and fix everything again—assuming you can remember where you made changes. So while `chcon` might be convenient, there's a better approach that actually keeps track of the context changes you are making.

In order to track your changes, you have to create a new entry in SELinux's internal file context policy table. We do this with "`semanage fcontext …`":

```
# semanage fcontext -a -t httpd_sys_content_t '/docroot(/.*)?'
# semanage fcontext -l | grep docroot
/docroot(/.*)?  all files  system_u:object_r:httpd_sys_content_t:s0
```

The first command above adds ("`-a`") a file context entry for '`/docroot(/.*?)`'—a regular expression that matches the `/docroot` directory itself and everything underneath it. In this case, we're specifying the default type attribute ("`-t`") "`httpd_sys_content_t`" for this directory and everything underneath. Since we didn't specify the user attribute or the role portions of the label, these properties were set to sensible default values. You can see this when we use "`semanage fcontext -l`" to list the file context table entries. Note that if you make a mistake when you add a file context entry, you can delete it by using "`semanage fcontext -d …`"—you use exactly the same syntax and all the same arguments you used when you did the "`-a`" version to add the rule initially.

While "`semanage fcontext …`" establishes a default policy for a given directory, it doesn't actually change the existing context labels. The preferred method for changing the labels is the `restorecon` command, which looks at the defaults in the file context table and applies the appropriate labels to the directory you specify:

```
# restorecon -FRvv /docroot
restorecon reset /docroot context user_u:object_r:default_t:s0->
system_u:object_r:httpd_sys_content_t:s0
restorecon reset /docroot/index.txt context
user_u:object_r:default_t:s0->
system_u:object_r:httpd_sys_content_t:s0
```

Here, we're using the recursive ("`-R`") option to make sure we adjust the context labels for `/docroot` and all our content beneath it, and the verbose ("`-vv`") option so we can see the changes as they're being made. "`-F`" forces `restorecon` to change the user and role portions of the context along with the type. We can confirm that our changes fixed the problem by attempting to access documents from our web server. You'll find that our changes fix the issues we were having.

One more useful tidbit: Suppose you've been making massive filesystem changes and you want to make sure that the file contexts on all directories reflect the default settings in the file context table. This often comes up after you've done a full restore of your filesystem, because many backup utilities may not capture the SELinux context information associated with files and directories. If you "`touch /.autorelabel`" and then reboot the system, then during the boot process, the system will do a "`restorecon -R -F`" on all local filesystems. Doing this during a reboot is recommended because it will be done very early in the boot process before any of the normal OS daemons have started—changing contexts on files and directories used by a process while that process is running can yield weird behavior. Note that doing a `restorecon` on the entire filesystem can take several minutes, so this isn't something you want to do all the time (at least if you value your uptime numbers).

ALTERNATE PORT? HAH! NO!

The scenario:

- You want a dev server on port 8888/tcp
- Change `httpd.conf` and restart
- SELinux terminates your process

The fix:

- Figure out context for `httpd` ports
- Add your new port to this context

Now let's look at how SELinux controls access to socket and port objects. Suppose you decided you wanted to run your web server on port 8888/tcp. After updating your `httpd.conf`, you attempt to restart the server and it gets killed due to an SELinux policy violation (again confirmed via the `audit.log` file).

```
# systemctl restart httpd
Job for httpd.service failed because the control process exited …
See "systemctl status httpd.service" and "journalctl -xe" for details.
# systemctl status httpd -l
…
Jul 09 14:42:08 localhost.localdomain httpd[4603]: (13)Permission
denied: AH00072: make_sock: could not bind to address 0.0.0.0:8888
# ausearch -c httpd
…
----
time->Fri Jul  9 14:42:08 2021
…
type=AVC msg=audit(1625856128.906:278): avc:  denied  { name_bind } for
pid=4603 comm="httpd" src=8888 scontext=system_u:system_r:httpd_t:s0
tcontext=system_u:object_r:unreserved_port_t:s0 tclass=tcp_socket
permissive=0
```

As was the case with our earlier docroot example, the web server worked fine when it was bound to port 80/tcp. So, there must be some SELinux context associated with 80/tcp that allows the web server to bind to that port. Clearly, this context is not associated with 8888/tcp by default. Can we change that? Of course!

DEALING WITH PORT CONTEXTS

```
# semanage port -l | grep ' 80,'
http_port_t   tcp   80, 81, 443, 488, 8008, 8009, 8443, 9000
# semanage port -a -t http_port_t -p tcp 8888
# semanage port -l | grep ' 8888,'
http_port_t   tcp   8888, 80, 81, 443, 488, 8008, 8009, 8443, 9000
```

Hilarity ensues when trying to reduce the default port list…

It turns out that we also use the `semanage` command to control port contexts, and you can see from the example on the slide that things are not much different from when we were using it to manage file contexts.

First, we look for port 80 in the output of "`semanage port -l`" so we can figure out the context name we need to associate with our new port 8888/tcp. Note that I'm `grep`-ing for "*<space>*`80`*<comma>*" here so that I match only port 80 in the output, and not ports like 8880, etc.

Once we figure out that the context we want to use is `http_port_t`, we can use "`semanage port -a …`" to add our new port to the list of ports associated with this context. Frankly, I wish the "`semanage port`" command took a sane argument like "8888/tcp", but I wasn't consulted when the command was being created. So, you're left with the kind of backward command syntax you see in the example above.

We can confirm our changes by using "`semanage port -l`" again. And you'll find that the web server starts successfully now that 8888/tcp is in the `http_port_t list`.

If you were to later remove 8888/tcp from this context ("`semanage port -d -t http_port_t -p tcp 8888`") you would find that the web server would continue to run on this port *until* you forced the web server to restart. SELinux works by blocking system calls like `bind()` to a network port. As long as the web server does not have to `bind()` to the port, SELinux doesn't interfere. So simply removing a port from a particular context does not immediately remove access from an application that is already listening on that port.

You'll notice that there are a bunch of other port numbers associated with the `http_port_t` context. If you weren't ever planning on using those ports, you might try using "`semanage port -d …`" to remove those ports from the list, thus further tightening the policy constraints on your web server. And when you try to do that, you see the following:

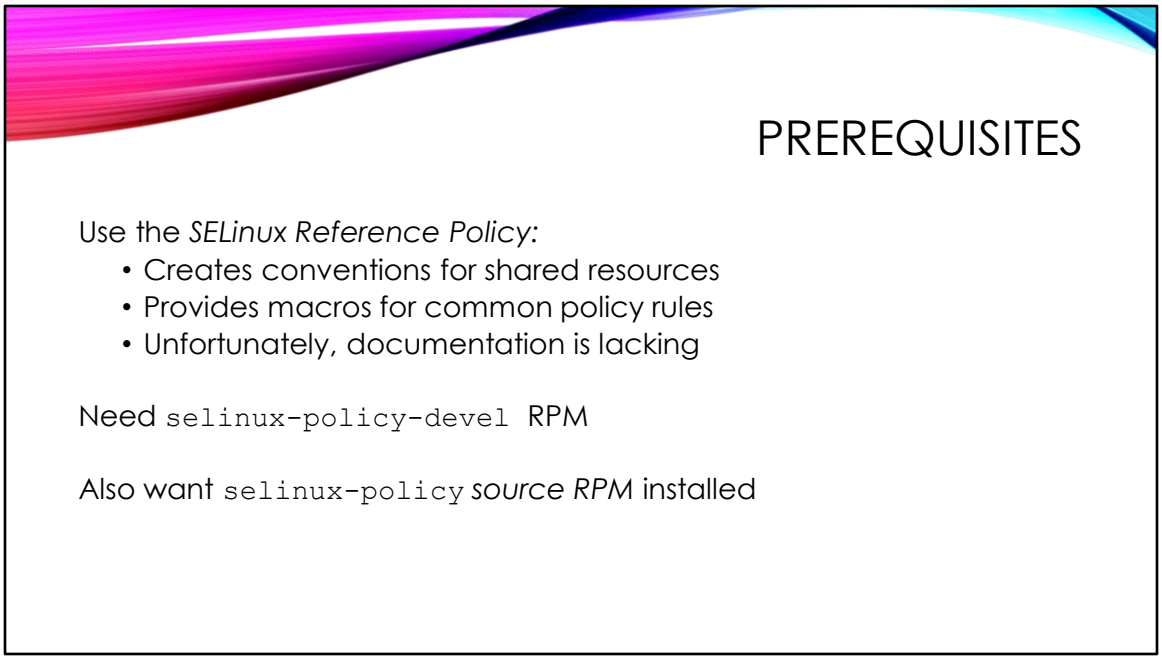
```
# semanage port -d -t http_port_t -p tcp 8009
ValueError: Port tcp/8009 is defined in policy, cannot be deleted
```

Yep, that's right. Removing these default ports would require modifying the source policy! Frankly, this is a bit of a bug.

Exercises are in HTML files in two places: on the downloaded course media, and in the home directory of the "lab" user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. Navigate to *…/Exercises/index.html* and open this HTML file
4. Click on the hyperlink which is the title of this exercise
5. Follow the instructions in the exercise

What if you had to had to create an entirely new policy for a brand-new application not currently covered by the default targeted policy? That gets a whole lot more interesting…



In the bad old days, you had to create raw SELinux policy from scratch. Imagine how awful that must have been. Consider a common shared service like Syslog—not only did you have to create a policy that allows Syslog to do all the things it needs to do (but not allow it to do too much) but then you had to negotiate access to all of the locations in the OS that are shared with other applications (`/etc`, `/var/log`, and so on). Plus, every other application that talks to Syslog would need to share the same set of allow rules to make that happen. Change any single piece and your changes would have to ripple out to the rest of your policy. It's a nightmare.

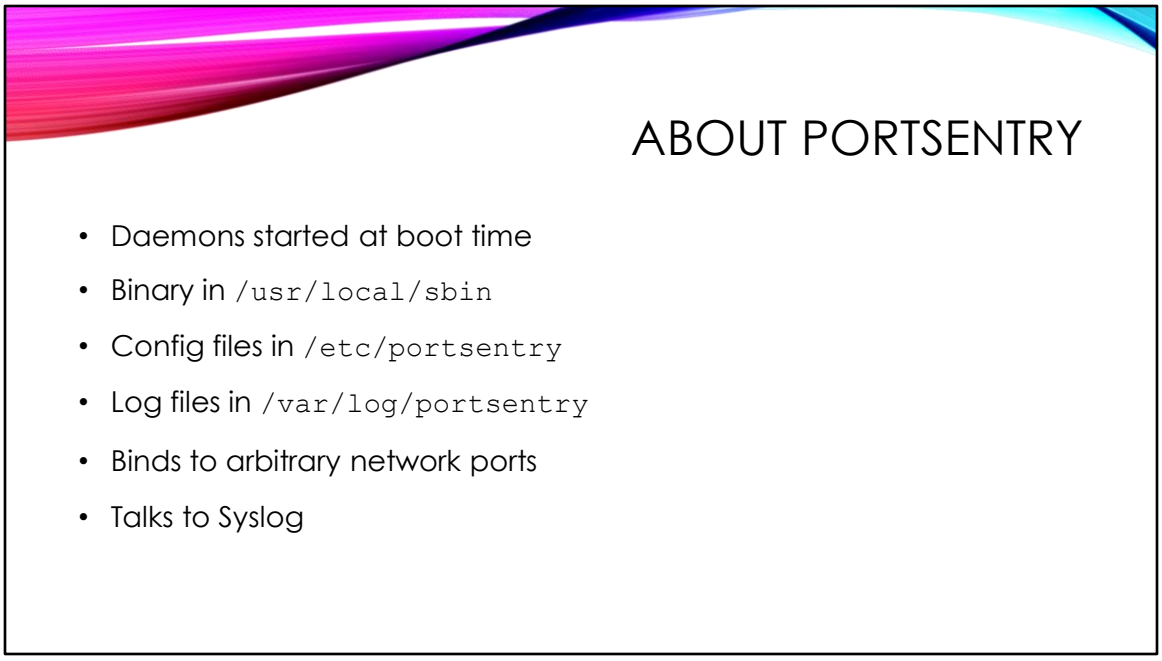
So, the idea for the Reference Policy was born. The Reference Policy establishes naming and usage conventions for shared resources in the operating system. It also creates macros for common access needs—sending log messages to the Syslog daemon is an example of one such macro. The bad news is that the Reference Policy is still actively being developed, and the documentation is limited.

So, we need to get a copy of the source. The current development site for the Reference Policy is *https://github.com/SELinuxProject/refpolicy* and you can find the download link there. But if you're using RHEL, then you're better off getting the source for the version that Red Hat used. I'll generally grab the source RPM from:

http://vault.centos.org/<osvers>/updates/Source/SPackages//selinux-policy-<vers>.src.rpm

When you install the source RPM, it unpacks itself into `/root/rpmbuild/SOURCES`. You'll find the files on this path in the virtual machine I gave you at the start of class. The file that actually contains the source code you want to look at is the `serefpolicy-`*`<vers>`*`.tgz` tar file. There's also `serefpolicy-contrib-`*`<vers>`*`.tgz` that contains sample policy files you can borrow ideas from. You can unpack these files anywhere in the filesystem that is convenient.

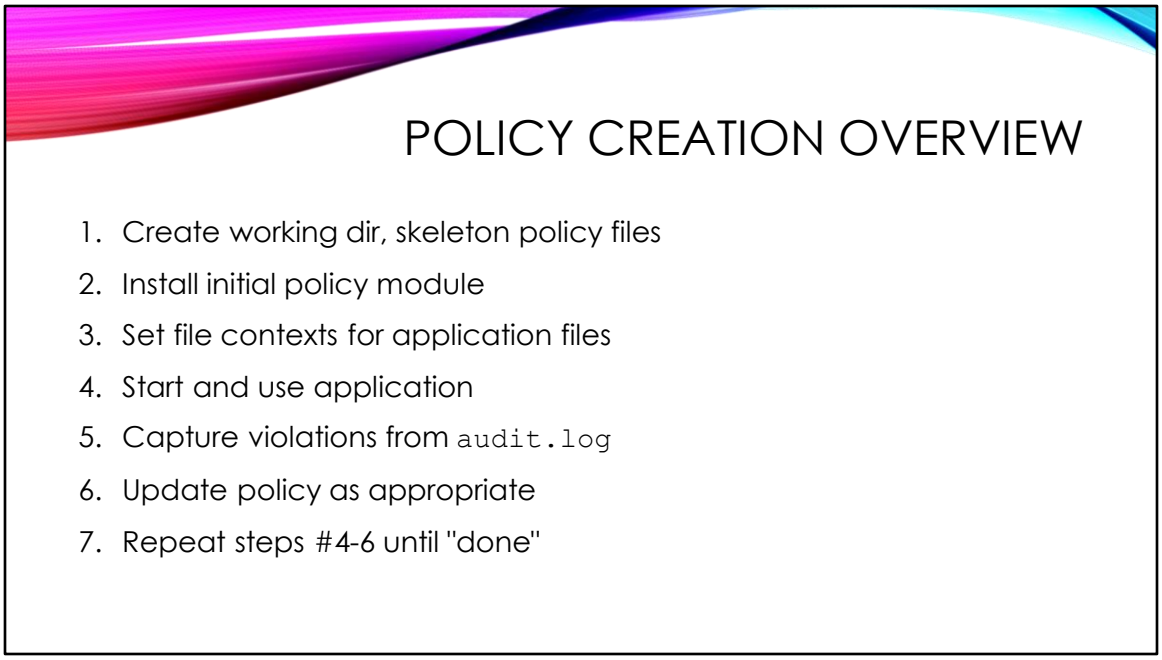
In addition to the files in the source RPM for your reference material, you also must have the `selinux-policy-devel` RPM installed. This RPM includes the sample policy files, but most importantly a `Makefile` that will make the whole compilation process a lot easier.



With the prerequisites out of the way, we're going to create a meaningful policy file for a real network service. For our example, we're going to create a policy for the PortSentry HIDS. PortSentry is an "interesting" example because it's started at boot time like other system services, has a configuration file in `/etc`, binds to network ports, talks to Syslog, and writes its own log files in a private directory. Basically, it covers all of the different kinds of access you're likely to run into for your own applications, and best of all, there's no existing policy module for it. Yay!

The SELinux policy creation process rewards those who follow standard file layout conventions, so we're going to do that. We'll put the `PortSentry` daemon binary in `/usr/local/sbin`, tell it to look for its configuration files under `/etc/portsentry`, and to write its logs to `/var/log/portsentry`. This will minimize the impact of the new software on the existing SELinux policy and make our jobs a whole lot easier.

It also helps to understand the kinds of access your application is going to need before you start crafting policy. You won't be able to predict everything up front, but the more you can "front load" the policy creation process, the less time you'll spend iterating to your final policy. In this case, we know that PortSentry is going to need to talk to Syslog and that it's going to need access to some network ports. However, unlike most applications that will typically only bind to a single port, PortSentry's behavior is to bind to a large number of essentially arbitrary ports. This is going to make the task of crafting SELinux policy in this area a little bit abnormal for our PortSentry policy, at least as compared to the network access sections for other services.



Before we dive into creating our policy for PortSentry, let's talk about the general process of developing a new policy module. First, you need to create a working directory where you'll be developing your policy file. This directory will have a copy of that `Makefile` from the `selinux-policy` RPM that I mentioned earlier, and it's also where you'll have the various input files you'll be compiling into policy.

You typically start with a minimal policy file where you define the kinds of access you know the application will need. That basic policy gets compiled and loaded into our policy database. Once the baseline policy is loaded, you monitor your `audit.log` file for violations and then use a tool called `audit2allow` to help generate additional rules for your policy based on the `audit.log` entries. You then compile an updated version of your policy module, load it, and repeat the process. You know you have a working policy when you stop seeing alerts in the `audit.log` file.

In case it wasn't already obvious, this is normally a process you would do on a non-production system with SELinux in Permissive mode. Permissive mode means that SELinux will allow the application to continue running so you see everything it's trying to do to the system and thus can generate the appropriate rules. Another reason to use a non-production system is that there are points where you're going to want to reboot the machine to both make sure the application is running in the correct context as well as to understand how the application behaves when the system is coming up.

Don't worry. Once you've figured out the policy in your test environment, it can just be installed directly on your production systems (assuming your test environment mirrors your production environment closely).

INITIAL SETUP

```
# mkdir -p /root/selinux/portsentry
# cd /root/selinux/portsentry/
# ln -s /usr/share/selinux/devel/Makefile .
# cp ~lab/portsentry/initial-policy/* .
# ls
Makefile  portsentry.fc  portsentry.if  portsentry.te
# make
Compiling targeted portsentry module
Creating targeted portsentry.pp policy package
rm tmp/portsentry.mod tmp/portsentry.mod.fc
```

Your working directory can live anywhere in the filesystem you want. I generally keep mine under `/root/selinux`. If you're doing a lot of policy development at your site, consider using a source code control system like Git.

Once you have your working directory, copy the `/usr/share/selinux/devel/Makefile` into it, or make a link to this file as I'm doing in the example above. You'll also need a copy of your initial policy files. I've provided some sample policy files in your VMware image and on your course CD, so you can just start with those. We'll look at these files in a lot more detail on the next slide.

When you're ready to compile your module, just run `make`. The `Makefile` takes care of the rest. The `portsentry.pp` file is the compiled "policy package" that we will load into our SELinux policy database.

SELinux policy module "packages" are generally created from three different files. The `*.te` file is the main file where you'll put your policy rules. The `*.if` file is where you're supposed to put any macros that you create specifically for your own policy. You could put these in the `*.te` file if you wanted, but it keeps things cleaner to put the macro definitions someplace else.

Finally, the `*.fc` file is used to hold file context definitions that apply to your application. These look a lot like the configuration we did earlier with "`semanage fcontext …`". The advantage to having these context definitions compiled into your policy package is that they will be automatically loaded into the file context table when you load the policy package and you won't have to muck around with `semanage` yourself. This is one of the reasons the module package format was created in the first place.

The next several pages will cover the contents of our initial straw-man policy files and introduce you to some of the common Reference Policy macros and other syntax elements…

First, let's look at our initial `portsentry.te` file:

```
policy_module(portsentry, 1.0.0)

#### Declarations

type portsentry_t;
type portsentry_exec_t;
init_daemon_domain(portsentry_t, portsentry_exec_t)

type portsentry_etc_t;
files_config_file(portsentry_etc_t)

type portsentry_log_t;
logging_log_file(portsentry_log_t);

# You need these lines if you're using an older init-based
Linux
#type portsentry_initrc_exec_t;
#init_script_file(portsentry_initrc_exec_t)

#### Policy

files_search_etc(portsentry_t)
files_search_var_log(portsentry_t)

logging_send_syslog_msg(portsentry_t)

miscfiles_read_localization(portsentry_t)
```

First, we use the `policy_module()` macro to declare our policy name and version number. Then we declare a number of context types that will be used by our application. In general, the prefix for all of the types you define should match the policy name from your `policy_module()` declaration—`portsentry_*` in our case. The extra suffixes like `*_exec_t`, `*_etc_t`, and `*_log_t` are also conventions used throughout the Reference Policy (other common types include names like `*_lib_t`, `*_tmp_t`, and `*_content_t`).

For each new type we define, there will typically be some associated macro declarations. For example, `init_daemon_domain()` is commonly used in policy files for daemons that are started at boot time. This macro defines (among other things) a *context transition* that happens whenever `systemd` or `init` runs an executable that belongs to our `portsentry_exec_t` context. The newly started process will be put into the `portsentry_t` context rather than inheriting the default context from the boot system. This transition is important so that we can write policy rules that only apply to our `portsentry_t` processes and not any other processes running on the system.

Other macros that are commonly used with our type declarations include `files_config_file()` and `logging_log_file()`, which generate the appropriate type declarations in your policy file for different types of files. It's important to note that these declarations don't actually grant any kind of access to these types of files; they merely handle the grunt work of properly declaring the file types. We'll get to access controls in later versions of our policy.

We know our configuration files are going to live under `/etc` and our log files will be written under `/var/log`. In order to be able to even access files in these directories, we need SELinux to give us access to them. That's the purpose of the `files_search_*()` macros. `files_search_etc()` expands to an allow rule that permits read-only access to the `/etc` directory for and programs running in `portsentry_t` context. `files_search_var_log()` is actually not a pre-defined macro in the Reference Policy—we'll be creating it in our `portsentry.if`, which I'll be showing you in a moment.

Finally, we end with some macros that will be common to nearly every policy file you write. You'll use `logging_send_syslog_msg()` in the policy file for any application that needs to communicate via Syslog. Similarly, most applications need to access the various language locale files in the system, so you'll want to use `miscfiles_read_localization()`.

I mentioned we needed to define our `files_search_var_log()` macro in `portsentry.if`, so let's take a look at that file next:

```
########################################
## <summary>
##      Search the /var/log directory.
## </summary>
## <param name="domain">
##      <summary>
##      Domain allowed access.
##      </summary>
## </param>
#
interface(`files_search_var_log',`
        gen_require(`
                type var_t, var_log_t;
        ')

        allow $1 { var_t var_log_t }:dir search_dir_perms;
')
```

To be honest, I created this macro by copying another similar macro called `files_search_var_lib()` and just replacing "`lib`" with "`log`". Unless you're a whiz with `m4`, I recommend doing the same thing.

Now `gen_require()` is a macro that generates a "`require { … }`" block to declare the types we're going to be using in our policy rules. Then we have an `allow` rule with some expansions: `$1` corresponds to the argument we give to the macro (`portsentry_t` in our case), and `search_dir_perms` is a macro defined elsewhere in the Reference Policy that corresponds to a list of access permissions. After being run through `m4`, the one line `files_search_var_log(portsentry_t)` gets expanded to the following policy rules:

```
require {
    type var_t, var_log_t;
}
allow portsentry_t { var_t var_log_t }:dir { getattr search };
```

That's about the least complicated macro you're likely to run into, but it gives you a flavor of how the macros are compiled into SELinux policy statements.

Finally, we have our `portsentry.fc` file:

```
/usr/local/sbin/portsentry          --
          gen_context(system_u:object_r:portsentry_exec_t,s0)
/etc/portsentry(/.*)?
           gen_context(system_u:object_r:portsentry_etc_t,s0)
/var/log/portsentry(/.*)?
           gen_context(system_u:object_r:portsentry_log_t,s0)
# You need this line if you're using an older init-based Linux
#/etc/rc\.d/init\.d/portsentry                    --
   gen_context(system_u:object_r:portsentry_initrc_exec_t,s0)
```

In the interest of readability, I was forced to introduce line breaks in the middle of each line. In practice, the `gen_context()` macros would appear as the third column in each line above—in other words, the actual `portsentry.fc` file is only five lines long.

Starting from the left-hand side of each line, the first column looks a lot like the file specification regular expressions we were using "`semanage fcontext …`" earlier. In the rightmost column, we use `gen_context()` to define what context we want the files and directories to have. Notice that has to fully declare the user context and the role along with the type. The "`,s0`" at the end of each context label is the *security level* as used by MLS. In the default targeted policy, everything is declared at "`s0`".

The middle column in the *.fc file syntax is optional and specifies the file type. For example, the first two entries use "--" to specify that the objects being referred to are regular files. "-d" would indicate a directory, "-c" a device file and so on. In two rules, we're using wildcards to specify a directory and all files underneath that directory, so using "--" or "-d" wouldn't work here because it wouldn't cover all cases. In these situations, just leave the second column blank and your rule will apply to all object types that match the given path specifier.

At this point, we've got some minimal policy description files, albeit ones that don't really grant our `portsentry_t` processes much in the way of access privileges. But it's enough to get us started. So, we type `make` in our build directory, and the `selinux-policy Makefile` automatically generates our `portsentry.pp` file for us. The next slide covers loading this policy package and some other implementation details.

LOAD MODULE, SET CONTEXTS

```
# semodule -i portsentry.pp
# semodule -l | grep portsentry
portsentry
# semanage fcontext -l | grep portsentry
/etc/portsentry(/.*)?                              all files
       system_u:object_r:portsentry_etc_t:s0
/etc/rc\.d/init\.d/portsentry                      regular file
       system_u:object_r:portsentry_initrc_exec_t:s0
/usr/local/sbin/portsentry                         regular file
       system_u:object_r:portsentry_exec_t:s0
/var/log/portsentry(/.*)?                          all files
       system_u:object_r:portsentry_log_t:s0
# restorecon -FR /etc/portsentry /var/log/portsentry \
       /usr/local/sbin/portsentry
```

Having generated our initial `portsentry.pp` file, we use "`semodule -i …`" to load it. We can confirm that the install worked with "`semodule -l`". We can also confirm that the file context definitions from our `portsentry.fc` file was properly loaded using "`semanage fcontext -l`".

However, while the file context table has been updated, the labels on our files and directories have not actually been updated. We have to run `restorecon` as shown on the slide above in order to set our file contexts properly.

Great, we've now got our initial policy loaded and our file contexts set. Again, the plan is to run the app on a system in Permissive mode and capture policy violations from the `audit.log` file. We're going to use a tool called `audit2allow` to help us generate the policy rules we're missing based on the policy violations recorded in the `audit.log`.

A couple of hints before you begin:

1. Create a boot configuration and start the process as it would normally start at boot time—in other words, reboot the system to start the process. If you just run the process from the command line, it may inherit the user context and role from your interactive session rather than using the user context and role that it will have in its production deployment. This difference will be reflected in the alerts in the `audit.log` file and will make it harder when you're trying to generate policy via `audit2allow`.

2. Just before you reboot the system, tell `auditd` to start a new `audit.log` file. You can do this by sending the `auditd` process a `SIGUSR1` signal via "`pkill -USR1 auditd`". If you start with a fresh `audit.log` file each time, then you won't waste time recreating policy statements from audit.log entries that you've already seen.

After you reboot the system and your process has been started, then go ahead and use the application as normal. In the case of PortSentry, this means hitting the system with some port scans to trigger PortSentry's logging behavior (and possibly other actions). You should also exercise the application by subjecting it to at least one more reboot cycle. There may be behaviors that you get when the application is shutting down and then coming back up that you may not see at any other time.

AUDIT2ALLOW 2 THE RESCUE!

```
# grep type=AVC /var/log/audit/audit.log |
    grep portsentry | audit2allow -m portsentry

module portsentry 1.0;

require {
   type portsentry_etc_t;
   type portsentry_t;
   …
}

#============= portsentry_t ==============
allow portsentry_t dhcpd_port_t:udp_socket name_bind;
…
```

The good news is that the SELinux packages include a little program called `audit2allow` that simplifies policy creation. You just feed your `audit.log` entries into `audit2allow` and it kicks out SELinux policy that would allow the access that's currently preventing your app from functioning. Obviously, you'll want to review the output of `audit2allow` and make sure the rules are not too permissive, but the tool can really help jump-start our policy.

When running `audit2allow`, you must specify a policy module name with "`-m`". You can see that `audit2allow` uses this information to generate a module header at the beginning of the output. In reality, we don't care about the first part of the `audit2allow` output where the module header and "`require`" block sit. What we're really interested in are the "`allow`" rules that come after the comment—these are the rules we will need to update our policy with.

AND THEN THE PAIN…

`audit2allow` has no clue about Reference Policy macros

So, you get to translate manually … yay!

Having a copy of the Reference Policy sources pays off …

Unfortunately, `audit2allow` dumps out raw SELinux policy rules. It has no clue about Reference Policy macros. You could insert the raw rules into the PortSentry policy file we're creating, but in the long term, that would end up being unmaintainable.

So, the bad news here is that you have to become an expert at translating raw SELinux policy statements back into Reference Policy macros. The good news is that there are some basic patterns that happen over and over again. So, after you've done this a few times, it gets to be pretty rote. But those first few times, you're really going to need a copy of the Reference Policy source to work with.

Don't panic! I'm going to work through this example with you to introduce you to how this process works. Stay with me …

REFERENCE POLICY TRANSLATION

Find clusters of related rules in output:

```
allow portsentry_t portsentry_etc_t:dir search;
allow portsentry_t portsentry_etc_t:file { getattr open read };
allow portsentry_t portsentry_log_t:dir { add_name search write };
allow portsentry_t portsentry_log_t:file { create open read write };
```

Two choices:

- Steal code from existing policies
- Hunt around with `find`/`grep`

The first hint is to break the process down into manageable chunks. When you look at our sample `audit2allow` output, it seems like a huge pile of confusing gibberish. But if you stare at it for a while, you'll start seeing some patterns that let you group some of the rules together in clumps like I've done below:

```
allow portsentry_t portsentry_etc_t:dir search;
allow portsentry_t portsentry_etc_t:file { read getattr open };
allow portsentry_t portsentry_log_t:dir { write search add_name };
allow portsentry_t portsentry_log_t:file { write read create open };

allow portsentry_t node_t:tcp_socket node_bind;
allow portsentry_t node_t:udp_socket node_bind;

allow portsentry_t dhcpd_port_t:udp_socket name_bind;
allow portsentry_t echo_port_t:tcp_socket name_bind;
allow portsentry_t echo_port_t:udp_socket name_bind;
...

allow portsentry_t self:capability { net_raw net_bind_service };
allow portsentry_t self:rawip_socket create;
allow portsentry_t self:tcp_socket { bind create listen };
allow portsentry_t self:udp_socket { bind create };
```

First, you see some rules related to file and directory access for objects in the `portsentry_etc_t` and `portsentry_log_t` contexts. This is where the daemon is trying to read its configuration files and write to its logs. Then there are some `node_bind` and `name_bind` rules (many of which I didn't show here in order to save space). Finally, there are some miscellaneous rules related to (raw) socket access. The trick is to focus on one chunk at a time in order to break the rule translation process up into manageable pieces.

Clearly, the `portsentry_etc_t` rules are related to the process accessing its configuration files under `/etc/portsentry`. There are lots of other daemons that have configuration files under `/etc` and many of these programs have SELinux policy files already written. So hopefully we can just steal some ideas from those pre-existing policy files.

Let's unpack the tarballs from the directory where we installed the source RPM and look around:

```
# cd /root/rpmbuild/SOURCES
# for f in selinux-policy-*.tar.gz; do tar zxf $f; done
# ls -d selinux-policy-*
selinux-policy-55f4df9.tar.gz
selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49
selinux-policy-contrib-5a34aed.tar.gz
selinux-policy-contrib-5a34aedf6563624d8543cbc708ba2a29be508872
# ls selinux-policy-contrib-5a34aedf6563624d8543…/*.te | wc -l
455
```

If you look in the `selinux-policy-contrib-`*<vers>* directory, you'll lots of sample policy files.

Your best bet is to start with one of the simpler policy files under the contrib directory. I've found that `soundserver.te` is pretty useful for examples:

```
# grep etc_t selinux-policy-contrib-*/soundserver.te
type soundd_etc_t alias etc_soundd_t;
files_config_file(soundd_etc_t)
read_files_pattern(soundd_t, soundd_etc_t, soundd_etc_t)
read_lnk_files_pattern(soundd_t, soundd_etc_t, soundd_etc_t)
```

The first two lines of output are very similar to declarations we already have in our basic PortSentry policy file. But the third line could be what we're looking for. However, we need to go looking for where `read_files_pattern` is defined in the core Reference Policy so that we can understand exactly what this macro does. Time for some command-line kung fu:

```
# cd /root/rpmbuild/SOURCES/selinux-policy-<vers>
# grep -rl read_files_pattern *
…
policy/modules/system/sysnetwork.te
policy/modules/system/systemd.if
policy/modules/system/systemd.te
policy/modules/system/udev.if
policy/modules/system/userdomain.if
policy/modules/system/userdomain.te
policy/support/file_patterns.spt
```

It turns out that the last file, `policy/support/file_patterns.spt` is the file we want. Here's the declaration for `read_files_pattern`:

```
define(`read_files_pattern',`
        allow $1 $2:dir search_dir_perms;
        allow $1 $3:file read_file_perms;
')
```

This is starting to look a little bit like the "`allow`" rules in our `audit2allow` output, but `search_dir_perms` and `read_file_perms` are themselves reference policy macros that are defined in a different file. If you do another round of "`grep –rl read_file_perms`", you'll end up looking at `policy/support/obj_perm_sets.spt`. In the middle of this file, you'll find:

```
#
# Directory (dir)
#
define(`getattr_dir_perms',`{ getattr }')
define(`setattr_dir_perms',`{ setattr }')
define(`search_dir_perms',`{ getattr search open }')
…

#
# Regular file (file)
#
define(`getattr_file_perms',`{ getattr }')
define(`setattr_file_perms',`{ setattr }')
define(`read_file_perms',`{ getattr open read lock ioctl }')
…
```

Great! We seem to have found the macros we're searching for, and many more. Putting the macros from the two files together, it looks like `read_files_pattern(portsentry_t, portsentry_etc_t, portsentry_etc_t)` would expand to two rules that look like this:

```
allow portsentry_t portsentry_etc_t:dir { getattr search open };
allow portsentry_t portsentry_etc_t:file { getattr open read … };
```

This seems to be a little bit more access than the rules we dumped out via `audit2allow` are saying we need. On the other hand, using the standard Reference Policy macros is a good idea for maintainability, so let's go with `read_files_pattern` here.

If we try and match up our `portsentry_log_t` rules from `audit2allow` against the macros in `obj_perm_sets.spt`, it looks like we need `add_entry_dir_perms` (the smallest permission set that includes `write` and `add_name` capabilities) plus `read_file_perms` and `write_file_perms`. `rw_file_perms` is close but doesn't include `create`. `manage_file_perms` is a little more access than we need. Unfortunately, there's no macro in `file_patterns.spt` that comes close to matching what we need, so instead, I'm going to make a new macro in `portsentry.if` :

```
interface(`rwcreate_files_pattern',`
        allow $1 $2:dir search_dir_perms;
        allow $1 $3:file read_file_perms;
        allow $1 $3:file write_file_perms;
')
```

Note that I have to use "`interface(…)`" to define the macro in `portsentry.if` rather than "`define(…)`" as the Reference Policy does in `file_patterns.spt`. This is due to the different ways in which the two files are compiled into policy.

Now we can update our `portsentry.te` file with the following macros:

```
read_files_pattern(portsentry_t, portsentry_etc_t, portsentry_etc_t)
rwcreate_files_pattern(portsentry_t, portsentry_log_t, portsentry_log_t)
```

So much for the file access rules; now let's tackle all those `name_bind` and `node_bind` entries. There are a lot of these but they're all generally the same, just for different ports. Let's try searching through our policy source directory for `name_bind` and see what turns up:

```
# grep -rl name_bind *
policy/flask/access_vectors
policy/mls
policy/modules/kernel/corenetwork.if.in
policy/modules/kernel/corenetwork.if.m4
policy/modules/kernel/corenetwork.te.in
```

Those …`/kernel/corenetwork.if.*` files look promising. If you scan through those files, you find lots of macros referring to `name_bind` and `node_bind`. But remember, we need to allow PortSentry to bind to pretty much any port it wants to, so the macros like `corenet_tcp_bind_all_ports()` seem most like what we need. Since we apparently need both `name_bind` and `node_bind` access to both TCP and UDP ports, the following collection of macros seems most apropos:

```
corenet_tcp_bind_all_ports(portsentry_t)
corenet_tcp_bind_all_nodes(portsentry_t)
corenet_udp_bind_all_ports(portsentry_t)
corenet_udp_bind_all_nodes(portsentry_t)
```

The only lines we haven't dealt with from the `audit2allow` output are the following:

```
allow portsentry_t self:capability { net_raw net_bind_service };
allow portsentry_t self:rawip_socket { read create };
allow portsentry_t self:tcp_socket { bind create listen };
allow portsentry_t self:udp_socket { bind create listen };
```

Turns out, there aren't really any macros in the Reference Policy to help us with these. There are certainly socket-related macros that are supersets of the "`{ bind create listen }`" and "`{ read create }`" access lists, but since we're allowing PortSentry to bind to any port on the system, we're probably better off only allowing the minimum access to these ports that we can. So, we're just going to shove these lines into our policy file verbatim.

Thus, our "final" `portsentry.te` file looks like this:

```
policy_module(portsentry, 1.0.1)

#### Declarations

type portsentry_t;
type portsentry_exec_t;
init_daemon_domain(portsentry_t, portsentry_exec_t)

type portsentry_etc_t;
files_config_file(portsentry_etc_t)

type portsentry_log_t;
logging_log_file(portsentry_log_t);

# You need these lines if you're using an older init-based Linux
#type portsentry_initrc_exec_t;
#init_script_file(portsentry_initrc_exec_t)

#### Policy

files_search_etc(portsentry_t)
files_search_var_log(portsentry_t)

read_files_pattern(portsentry_t, portsentry_etc_t,
portsentry_etc_t)
rwcreate_files_pattern(portsentry_t, portsentry_log_t,
portsentry_log_t)

corenet_tcp_bind_all_ports(portsentry_t)
corenet_tcp_bind_all_nodes(portsentry_t)
corenet_udp_bind_all_ports(portsentry_t)
corenet_udp_bind_all_nodes(portsentry_t)

allow portsentry_t self:capability { net_raw net_bind_service };
allow portsentry_t self:rawip_socket { read create };
allow portsentry_t self:tcp_socket { bind create listen };
allow portsentry_t self:udp_socket { bind create listen };

logging_send_syslog_msg(portsentry_t)

miscfiles_read_localization(portsentry_t)
```

NEW RULES, NOW WHAT?

Update your `*.te` file (change policy version number!)

Build and load new policy

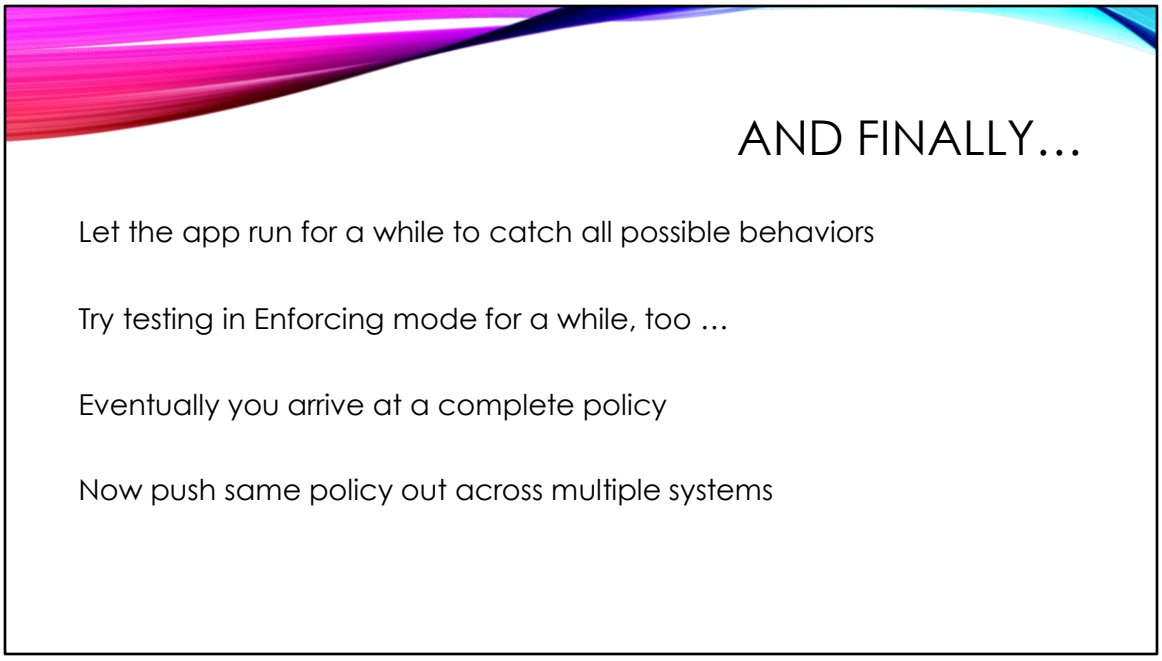
Start new `audit.log`, reboot system

Exercise daemon, check `audit.log`…

Keep going until `audit.log` warnings are gone

Awesome! We've got a new `portsentry.te` file to try out, this one labeled with version marker 1.0.1. Having saved the changes to the file in our build directory, we run `make` again and install the new `portsentry.pp` file. "`semodule -l`" should show that the installed module version is now 1.0.1.

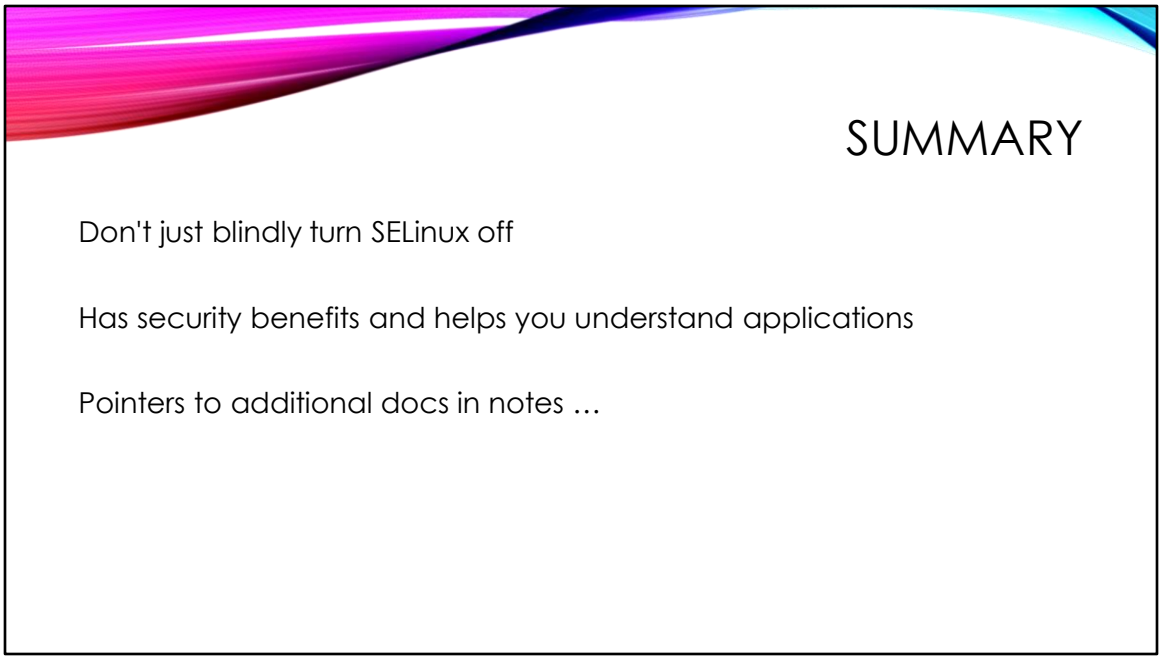
At this point, it's really a question of "lather, rinse, repeat." Start a new `audit.log` file with "`pkill -USR1 auditd`" and reboot the machine. Fire some more port scans at the box to let PortSentry do its thing. Monitor your `audit.log` for any new warnings. If any warnings show up, follow the same procedure we just went through to find the appropriate macros to update your policy file, and so on.



In general, it's also probably a good idea to let the application run for a while, just in case there are occasional behaviors of the application—weekly cleanup tasks, for example—that you didn't catch during your testing. It's also a good idea at some point during this "burn in" period to put your test system into enforcing mode as a final check that your policy is working as expected.

Eventually, you'll have high confidence that your policy is correct and complete. At this point, you can push the policy out across your enterprise. Assuming your systems are running the same OS version, you should just be able to copy the `*.pp` file out to each machine and load it with "`semodule -i`". Don't forget to do a `restorecon` to set the right contexts on the files and directories associated with the application.

If you have different OS versions running around, you'll need to recompile a `*.pp` file for each different version you're running—but again using the Reference Policy macros should help insulate you from changes between OS revisions.

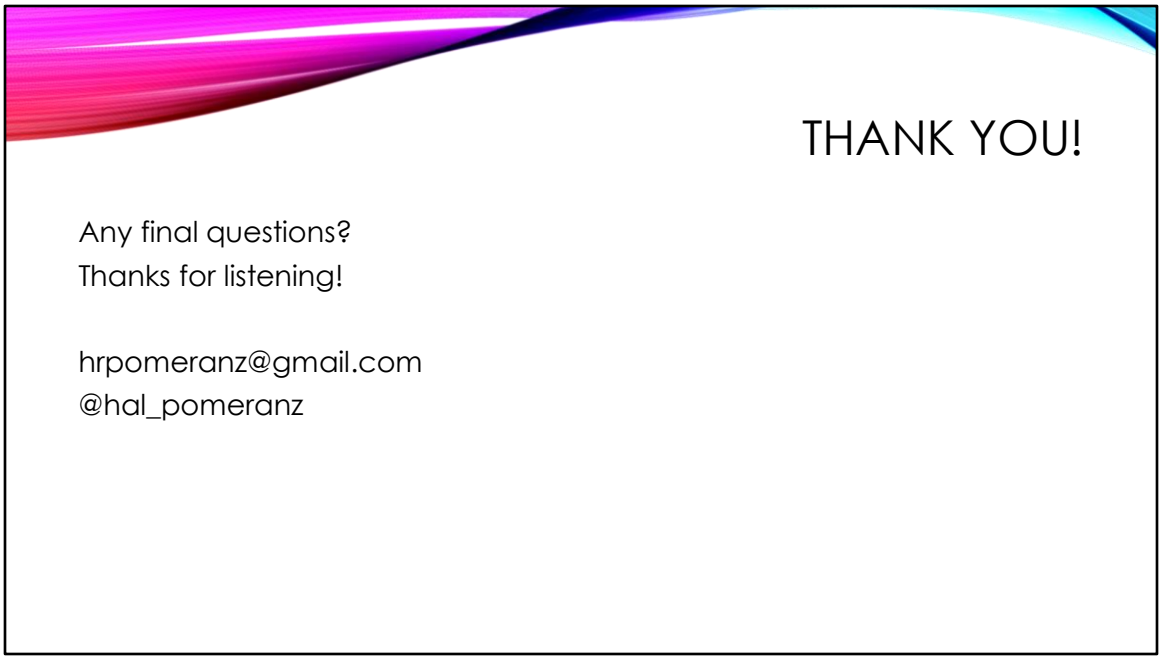


Is enabling SELinux the right choice for your environment? All too often, sites don't make an informed choice on this issue—they just disable SELinux immediately because they don't understand it. Hopefully, the material in this course will clear up some of the fear, uncertainty, and doubt around SELinux and let you make an honest, intelligent decision for your organization. Maybe I've even sparked your interest in SELinux a bit.

SELinux does have the promise of providing much higher levels of security. But we're only going to work out the kinks in the default policy if people actually start using it. Creating a policy for an application also teaches you a lot about how that application interacts with the operating system.

More documentation on SELinux:

A good intro to SELinux: *http://www.billauer.co.il/selinux-policy-module-howto.html*
Excellent How-To from the CentOS project: *http://wiki.centos.org/HowTos/SELinux*
The Fedora Wiki: *https://fedoraproject.org/wiki/SELinux*
Reference Policy development site: *https://github.com/SELinuxProject/refpolicy/wiki*



I hope you learned a lot from this material and had some fun along the way.

If you have questions or feedback in the future, please don't hesitate to contact me:

Hal Pomeranz
hrpomeranz@gmail.com
@hal_Pomeranz

Download updates from *https://archive.org/details/HalSELinux*

Please support continued development of this material by taking one of my training classes or donating (US$20 is suggested) via PayPal (*paypal.me/halpomeranz*) or Patreon (*patreon.com/halpomeranz*).

Exercises are in HTML files in two places: on the downloaded course media, and in the home directory of the "lab" user in the lab virtual machine. These two locations contain identical copies of the exercise files—it just depends on whether you prefer to look at them from a web browser inside the lab VM (because you're working in full-screen mode) or from your host operating system.

1. Open your web browser
2. Select "File… Open…" from the browser menu (Ctrl-O is the usual keyboard shortcut)
3. Navigate to *…/Exercises/index.html* and open this HTML file
4. Click on the hyperlink which is the title of this exercise
5. Follow the instructions in the exercise